Outline for Today

• Objective:
  – Continue with “Classic Problems”
  – To introduce message passing
  – Unix process-oriented system calls
• Administrative details:
  – Start program 2!
  – Making up missed classes

Dealing with Deadlock

It can be prevented by breaking one of the prerequisite conditions:
• Mutually exclusive use of resources
  – Example: Allowing shared access to read-only files (readers/writers problem)
• Circular waiting
  – Example: Define an ordering on resources and acquire them in order
• Hold and wait
• No pre-emption

Circular Wait Condition

```
while (food available)
{
    if (me == 0) {P(fork[left(me)]);
P(fork[right(me)]);} 
else {P(fork[right(me)]); P(fork[left(me)]); }
    eat;
    V(fork[left(me)]); V(fork[right(me)]);
    think awhile;
}
```

5 Dining Philosophers
Hold and Wait Condition

while (food available)
{
    P(mutex);
    while (forks[me] != 2)
    {
        blocking[me] = true; V(mutex); P(sleepy[me]); P(mutex);
        forks[leftneighbor(me)]--; forks[rightneighbor(me)]--;
    }
    V(mutex);
    eat:
    P(mutex); forks[leftneighbor(me)] ++; forks[rightneighbor(me)]++;
    if (blocking[leftneighbor(me)]) blocking[leftneighbor(me)] = false;
    V(sleepy[leftneighbor(me)]);
    if (blocking[rightneighbor(me)]) blocking[rightneighbor(me)] = false;
    V(mutex);
    think awhile;
}

Starvation

The difference between deadlock and starvation is subtle:
- Once a set of processes are deadlocked, there is no future execution sequence that can get them out of it.
- In starvation, there does exist some execution sequence that is favorable to the starving process although there is no guarantee it will ever occur.
- Rollback and Retry solutions are prone to starvation.
- Continuous arrival of higher priority processes is another common starvation situation.

5 Dining Philosophers

5DP - Monitor Style

Boolean eating[5];
Lock forkMutex;
Condition forksAvail;
void PickupForks (int i) {
    forkMutex.Acquire();
    while ( eating[(i+1)%5] || eating[(i+4)%5] )
    {
        forksAvail.Wait(& forkMutex);
        eating[i] = false;
        forkMutex.Broadcast(&forkMutex);
        forksAvail.Wait(&forkMutex);
        eating[i] = true;
        forkMutex.Release();
    }
    forkMutex.Release();
}
Template for Philosopher

while (food available)
{
    PickupForks(me); /*pick up forks*/
    eat;
    PutdownForks(me); /*put down forks*/
    think awhile;
}

What about this?

while (food available)
{
    forkMutex.Acquire();
    while (forks[me] != 2) { blocking[me]=true;
        forkMutex.Release(); sleep(); forkMutex.Acquire(); } forks[leftneighbor(me)]--; forks[rightneighbor(me)]--;
    blocking[me]=false;
    forkMutex.Release();
    eat;
    forkMutex.Acquire(); forks[leftneighbor(me)] ++; forks[rightneighbor(me) ]++; if (blocking[leftneighbor(me)] || blocking[rightneighbor(me)])
        wakeup(); forkMutex.Release();
    think awhile;
}

Readers/Writers Problem

Synchronizing access to a file or data record in a database such that any number of threads requesting read-only access are allowed but only one thread requesting write access is allowed, excluding all readers.

Template for Readers/Writers

Reader()
{ while (true)
    { *request r access*/
        read
        *release r access*/
    }
}

Writer()
{ while (true)
    { *request w access*/
        write
        *release w access*/
    }
}
R/W - Monitor Style

Boolean busy = false;
int numReaders = 0;
Lock filesMutex;
Condition OKtoWrite, OKtoRead;

void startRead() {
    filesMutex.Acquire();
    while (busy)
        OKtoRead.Wait(&filesMutex);
    numReaders ++;
    filesMutex.Release();
}

void endRead() {
    filesMutex.Acquire();
    numReaders --;
    if (numReaders == 0)
        OKtoWrite.Signal(&filesMutex);
    filesMutex.Release();
}

Semaphore Solution with Writer Priority

int readCount = 0, writeCount = 0;
semaphore mutex1 = 1, mutex2 = 1;
semaphore readBlock = 1;
semaphore writePending = 1;
semaphore writeBlock = 1;

void startWrite() {
    filesMutex.Acquire();
    while (busy || numReaders != 0)
        OKtoWrite.Wait(&filesMutex);
    busy = true;
    filesMutex.Release();
}

void endWrite() {
    filesMutex.Acquire();
    busy = false;
    OKtoRead.Broadcast(&filesMutex);
    OKtoWrite.Signal(&filesMutex);
    filesMutex.Release();
}
Reader()
while (TRUE) {
    other stuff;
P(writePending);
P(readBlock);
P(mutex1);
readCount = readCount +1;
if( readCount == 1)
P(writeBlock);
V(mutex1); V( readBlock );
V(writePending);
access resource;
P(mutex1);
readCount = readCount - 1;
if(readCount == 0)
V(writeBlock);
V(mutex2);
}

Writer()
while(TRUE){
    other stuff;
P(mutex2);
writeCount = writeCount +1;
if ( writeCount == 1)
P(readBlock);
V(mutex2); P(writeBlock);
access resource;
V(writeBlock);
P(mutex2);
writeCount - writeCount - 1;
if ( writeCount == 0)
V(readBlock );
V(mutex2);
}

Assume the writePending semaphore was omitted in the solution just given. What would happen?

This is supposed to give writers priority. However, consider the following sequence:
Reader 1 arrives, executes thro’ P(readBlock);
Reader 1 executes P(mutex1);
Writer 1 arrives, waits at P(readBlock);
Reader 2 arrives, waits at P(readBlock);
Reader 1 executes V(mutex1); then V(readBlock);
Reader 2 may now proceed…wrong

Interprocess Communication - Messages
• Assume no explicit sharing of data elements in the address spaces of processes wishing to cooperate/communicate.
• Essence of message-passing is copying (although implementations may avoid actual copies whenever possible).
• Problem-solving with messages - has a feel of more active involvement by participants.

Issues
• System calls for sending and receiving messages with the OS(s) acting as courier.
  – Variations on exact semantics of primitives and in the definition of what comprises a message.
• Naming - direct (to/from pids), indirect (to distinct objects - e.g., mailboxes, ports, sockets)
  – How do unrelated processes “find” each other?
• Buffering - capacity and blocking semantics.
• Guarantees - in-order delivery? no lost messages?
Remote Procedure Call - RPC

• *Looks* like a nice familiar procedure call

```plaintext
P_0
result = foo(param);  

P_1
Receive
```

Remote Procedure Call - RPC

• *Looks* like a nice familiar procedure call

```plaintext
P_0
result = foo(param);

P_1
Receive
```

Remote Procedure Call - RPC

• *Looks* like a nice familiar procedure call

```plaintext
P_0
result = foo(param);  

P_1
Receive
```

Remote Procedure Call - RPC

• *Looks* like a nice familiar procedure call

```plaintext
P_0
result = foo(param);

P_1
Receive
```
Remote Procedure Call - RPC

- Looks like a nice familiar procedure call

P₀
result = foo(param);

P₁
Receive r = foo(param);
// actual call
Reply

P₀
returning r to P₁

Remote Procedure Call - RPC

- Looks like a nice familiar procedure call

P₀
result = foo(param);

P₁
Receive r = foo(param);
// actual call
Reply

Template for Philosopher

while (food available)
{
    PickupForks(me); /*pick up forks*/
    eat;
    PutdownForks(me); /*put down forks*/
    think awhile;
}

5DP via RPC

- Looks like a nice familiar procedure call

Philosopher₀
result = PickupForks (0);

Fork Server
Receive r = proc(param);
// explicit queuing when necessary
Reply
5 DP - Send/Receive Msg
Passing

Fork please?

Philosopher 0 (thinking)

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

Philosopher 0

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

Fork please?

Philosopher 0 (thinking)

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

Fork please?

Philosopher 0 (thinking)

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

Fork please?

Philosopher 0 (thinking)

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

I’ll ignore that request until I’m done

Philosopher 0 (thinking)

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

Fork please?

Philosopher 0 (thinking)

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

Umm. Oh yeah.

Philosopher 0

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4
Unix Process Model

- Simple and powerful primitives for process creation and initialization.
  - `fork` syscall creates a child process as (initially) a clone of the parent
  - parent program runs in child process to set it up for `exec`
  - child `can exit`, parent can `wait` for child to do so.
- Rich facilities for controlling processes by asynchronous signals.
  - notification of internal and/or external events to processes or groups
  - the look, feel, and power of interrupts and exceptions
  - default actions: stop process, kill process, dump core, no effect
  - user-level handlers

Unix Process Control

```
int pid;
int status = 0;
if (pid = fork()) {
    /* parent */
    ...
    pid = wait(&status);
} else {
    /* child */
    ...
    exit(status);
}
```

- Parent uses `wait` to sleep until the child exits; `wait` returns child pid and status.
- `Wait` variants allow wait on a specific child, or notification of stops and other signals.
- Child process passes status back to parent on `exit`, to report success/failure.

Child Discipline

- After a `fork`, the parent program has complete control over the behavior of its child.
- The child inherits its execution environment from the parent...but the parent `program` can change it.
  - sets bindings of file descriptors with `open`, `close`, `dup`
  - `pipe` sets up data channels between processes
- Parent program may cause the child to execute a different program, by calling `exec*` in the child context.
Exec, Execve, etc.

• Children should have lives of their own.
• Exec* "boots" the child with a different executable image.
  – parent program makes exec* syscall (in forked child context) to run a program in a new child process
  – exec* overlays child process with a new executable image
  – restarts in user mode at predetermined entry point (e.g., crt0)
  – no return to parent program (it’s gone)
  – arguments and environment variables passed in memory
  – file descriptors etc. are unchanged

Fork/Exit/Wait Example

OS resources
fork
parent
fork
child
wait exit
Child process starts as clone of parent: increment refcounts on shared resources.
Parent and child execute independently: memory states may diverge.
On exit, release memory and decrement refcounts on shared resources.
Child enters zombie state: process is dead and most resources are released, but process descriptor remains until parent reaps exit status via wait.
Parent sleeps in wait until child stops or exits.

Join Scenarios

• Several cases must be considered for join (e.g., exit/wait).
  – What if the child exits before the parent joins?
    • "Zombie" process object holds child status and stats.
  – What if the parent continues to run but never joins?
    • How not to fill up memory with zombie processes?
  – What if the parent exits before the child?
    • Orphans become children of init (process 1).
  – What if the parent can’t afford to get “stuck” on a join?
    • Asynchronous notification.

Unix Signals

• Signals notify processes of internal or external events.
  – the Unix software equivalent of interrupts/exceptions
  – only way to do something to a process “from the outside”
  – Unix systems define a small set of signal types
• Examples of signal generation:
  – keyboard 
    • ctrl-c and ctrl-z signal the foreground process
  – synchronous fault notifications, syscall errors
  – asynchronous notifications from other processes via kill
  – IPC events (SIGPIPE, SIGCHLD)
  – alarm notifications
Process Handling of Signals

1. Each signal type has a system-defined default action.
   - abort and dump core (SIGSEGV, SIGBUS, etc.)
   - ignore, stop, exit, continue

2. A process may choose to block (inhibit) or ignore some signal types.

3. The process may choose to catch some signal types by specifying a (user mode) handler procedure.
   - specify alternate signal stack for handler to run on
   - system passes interrupted context to handler
   - handler may munge and/or return to interrupted context

User’s View of Signals

```c
int alarmFlag = 0;
alarmHandler () {
    printf("An alarm clock signal was received\n");
    alarmFlag = 1;
}
main () {
    signal (SIGALRM, alarmHandler );
    alarm(3); printf("Alarm has been set\n");
    while (!alarmflag ) pause ();
    printf("Back from alarm signal handler\n");
}
```

User’s View of Signals II

```c
main () {
    int (*oldHandler) ();
    printf("I can be control-c’ed\n");
    sleep (3);
    oldHandler = signal (SIGINT, SIG_IGN);
    printf("I'm protected from control-c'\n");
    sleep(3);
    signal (SIGINT, oldHandler);
    printf("Back to normal\n");
    sleep(3); printf("bye\n");
}
```

Yet Another User’s View

```c
main(argc, argv){
    int pid;
    signal (SIGCHLD, childhandler );
    pid = fork ();
    if (pid == 0) /*child*/{ execvp (argv[2], & argv[2]); } else { sleep (5); printf("child too slow\n"); kill (pid, SIGINT); }
}
childhandler () { int childPid, childStatus ;
    childPid = wait (& childStatus);
    printf("child done in time\n");
    exit;
}
```
Files (& everything else)

- **Descriptors** are small unsigned integers used as handles to manipulate objects in the system, all of which resemble files.
- **open** with the name of a file returns a descriptor
- **read** and **write**, applied to a descriptor, operate at the current position of the file offset. **lseek** repositions it.
- Pipes are unnamed, unidirectional I/O stream created by **pipe**.
- Devices are special files, created by **mknod**, with **ioctl** used for parameters of specific device.
- Sockets introduce 3 forms of **sendmsg** and 3 forms of **recvmsg** syscalls.

```
Files System Calls

open files are named to by
an integer
file descriptor.

Standard descriptors (0, 1, 2) for
input, output, error messages
(stdin, stdout, stderr).

Pathnames may be relative
to process current directory.

Process does not specify
current file offset: the
system remembers it.

Process passes status back to
parent on exit, to report
success/failure.

Open files are named to by
an integer
file descriptor.

Standard descriptors (0, 1, 2) for
input, output, error messages
(stdin, stdout, stderr).

File Sharing Between
Parent/Child

main(int argc, char *argv[]) {
    char c;
    int fdrd, fdwt;
    if ((fdrd = open(argv[1], O_RDONLY)) == -1)
        exit(1);
    if ((fdwt = creat(argv[2], 0666)) == -1)
        exit(1);
    fork();
    for (;;) {
        if (read(fdrd, &c, 1) != 1)
            exit(0);
        write(fdwt, &c, 1);
    }
}"
```

Sharing Open File Instances

- Parent file descriptor is respected by child.
- When child reads the file, it maintains the file offset.
- In the child, the file offset started with the file descriptor passed by the parent.
- The parent can use `fork()` or `vfork()` to share the file.
Setting Up Pipelines

```c
int pfd[2] = {0, 0}; /* pfd[0] is read, pfd[1] is write */
in, out; /* pipeline entrance and exit */
pipe(pfd); /* create pipeline entrance */
out = pfd[0];
in = pfd[1];
/* loop to create a child and add it to the pipeline */
for (i = 1; i < procCount; i++) {
    out = setup_child(out);
}
/* pipeline is a producer/consumer bounded buffer */
write(in, ..., ...);
read(out, ..., ...);
```

### Setting Up a Child in a Pipeline

```c
int setup_child(int rfd) {
    int pfd[2] = {0, 0}; /* pfd[0] is read, pfd[1] is write */
in, wfd;
pipe(pfd); /* create right-hand pipe */
wfd = pfd[1];/* this child's write side */
if (fork()) { /* parent */
close(wfd); close(rfd);
} else { /* child */
close(pfd[0]); /* close far end of right pipe */
close(0); /*stdin*/ close(1); /*stdout*/
dup(rfd); /*takes fd 0*/ dup(wfd); /*takes fd 1*/
close(0); close(wfd);
...
}
return(pfd[0]);
```

### Sockets for Client-Server Message Passing

**Server**
1. Create a named socket
   ```c
   sfd = (socket(…))
   bind(sfd, …)
   listen(sfd, …)
   ```
2. Listen for clients
   ```c
   client(sfd, …)
   ```
3. Connection made and continue listening
   ```c
   cfd=accept(sfd, …)
   ```
4. Exchange data
   ```c
   write(cfd, …)
   ```
5. Done
   ```c
   close(cfd);
   ```

**Client**
1. Create unnamed socket and ask for connection
   ```c
   cfd=socket(…)
   err=connect(cfd, …)
   ```
2. Connection made
   ```c
   cfd=accept(sfd, …)
   ```
3. Exchange data
   ```c
   read(cfd, …)
   ```
4. Done
   ```c
   close(cfd);
   ```
End-to-End Argument

- Application-level correctness requires checking at the endpoints to ensure that the message exchange accomplished its purpose
  - Application semantics involved
  - Notification of successful delivery (UPS tracking) is not as good as a direct response (thank you note) from the other end.
- Reliability guarantees in the message-passing subsystem provide performance benefits (short-circuiting corrective measures).
  - Re-transmitting packet may save re-transferring whole file.