LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 634: Networks Programming

Lecture 04: Signals, Process Control, Pipes, and Daemon Processes

Instructor: Haidar M. Harmanani

Today's Agenda

- Managing Unix Processes
- Inter-Process Communication
- Posix Threads
- Java Threads

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Managing Unix Processes

System Calls

- A System Call is a call into the Unix kernel. Three important categories of syscalls:
  - Process management: fork , exec , wait , waitpid
  - Signals: Simple form of inter-process communication
  - File management
- They are called as if regular C function calls, but they are part of Unix (not C)

Programs vs. Processes

- Computers execute programs

- A process is one instance of a program, while it is executing
  - A process is simply a program in execution

- The same program can be executed multiple times in parallel
  - Somebody else may log on my computer and start that simulator too
  - These are several separate processes, executing the same program

- A process can only be created by another process
  - E.g., when we type a command, the Unix shell process will create a new process to execute it

- A process has a unique **P**rocess **ID**entier (**PID**)

Inside a Process

- A process is made of:
  - One executing program (i.e., a file containing the code to run)
  - PID: Process identifier (an integer)
  - Memory used to execute the program (text, data, heap, stack)
  - PC: Program counter
    - indicates where in the program the process currently is
  - A number of signal handlers
    - tells the program what to do when receiving signals
- A process can determine its own PID:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

- Or the PID of its parent:

```
pid_t getppid(void);
```

fork

- fork creates a near-clone of the calling process
  - Same values of variables
  - Same point of execution
  - Same open files, same current working directory
- Both processes run **concurrently**; we can't predict their order of execution
- The new process created by fork is the child of the original parent process

The fork() System Call [1/2]

- In Unix systems, there is exactly one way to create a new process:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- The child process is an exact copy of the parent
  - It is running the same program
  - Its memory area is an exact copy of the parent's memory
  - Signal handlers and file descriptors are copied too
  - Its program counter (PC) is at the same position within the program
    - I.e., just after the fork() call
- There is one way of distinguishing between the two processes:
  - fork() returns 0 to the child process
  - fork() returns the child's PID to the parent process
  - fork() returns -1 in case of error
- Most often, programs need to check the return value of fork().

The `fork()` System Call [2/2]

- Typically, processes diverge after fork() returns:

```
pid_t pid;
pid = fork();

if (pid<0) {perror("Fork error"); exit(1);}

if (pid == 0)/* child */
{
  printf("I am the child process\n");
  while (1) putchar("c");
}
else          /* parent */
{
  printf("I am the parent process\n");
  while(1) putchar("p");
}
```


The Fork of Death!

- You must be careful when using fork()!!
  - Very easy to create very harmful programs

```
while (1) {
   fork();
}
```

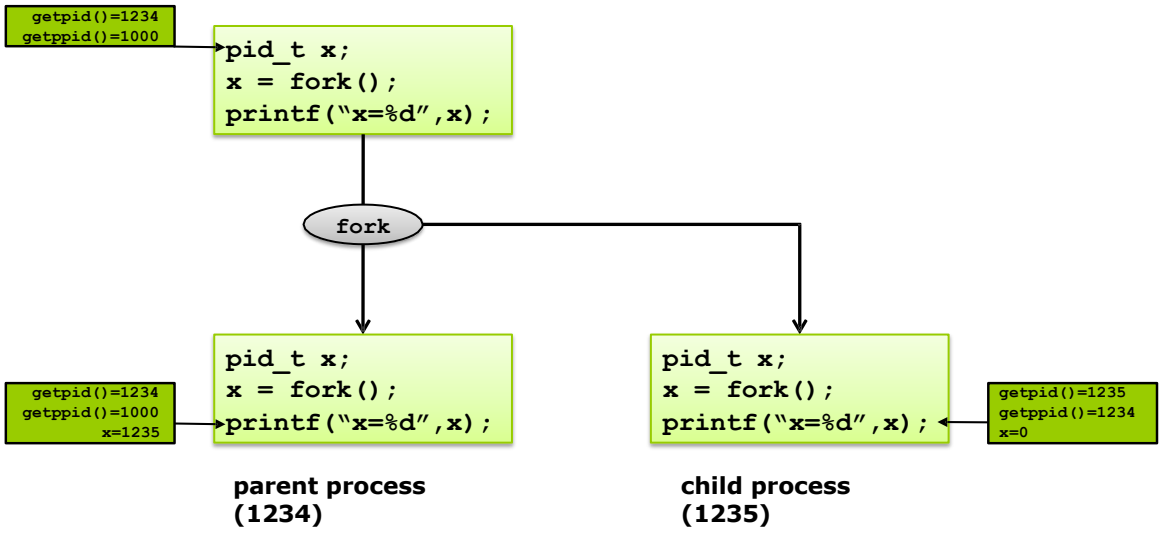
- Finite number of processes allowed
  - PIDs are 16-bit integers ➡️ maximum 65536 processes!

- Administrators typically take precautions
  - Limited quota on the number of processes per user
  - Try this at home ☺

fork() Return Value

- If the fork succeeds, it returns the newly-created child's PID to the parent, and 0 to the child
  - This differing return value allows us to have parent and child do differing things after the fork
- If fork fails, it returns -1 to the parent and sets errno
  - fork can fail if you run into per-user or system-wide process limits
  - PIDs are positive integers; -1 cannot be confused for a PID

fork() Return Value



getpid and getppid

- getpid returns PID of current process
- getppid returns PID of parent process

```
#include<unistd.h>
#include<stdio.h>

int main(void) {
pid_t child;
printf("I am %d\n", getpid());
child = fork();
if (child)
     printf("Parent is %d\n", getpid());
else {
     printf("Child is %d\n", getpid());
     printf("Child's parent is %d\n", getppid());
     }
return 0;
}
```


fork Variables

```
#include <unistd.h>
#include <stdio.h>
int main(void)
{
  int i = 5;
  pid_t child;
  child = fork();

  if (child) {
     i++;
     printf("%d\n", i);
  }
  else {
     i++;
  printf("%d\n", i);
  }
return 0;
}
```

fork Variables

```
#include <unistd.h>
#include <stdio.h>

int main(void) {
  int i;
  pid_t child;

  for (i = 0; i < 3; i++) {
     child = fork();
     if (child == 0)
          printf("Child %d saying hi\n", getpid());
  }
  return 0;
}
```


Zombie Children

- As with every other process, a child process terminates with an exit status
- This exit status is often of interest to the parent who created the child
- But what happens if the child process terminates before the parent can grab its exit status?
- The child turns into a zombie process
  - Minimal information about the process is retained
  - When the parent obtains the exit status, the zombie will get removed
- How does the parent accept the exit status?

Zombie Processes

- When a process terminates, it is not immediately removed from the system
  - The process which created it may be interested in its return value
- It gets to zombie state:
  - Its memory and resources are freed
  - It stays in the process table until its parent has received its termination status
  - If the parent has already finished, it is adopted by process 1 (init)
- Zombie processes must be dealt with!
  - Otherwise we eventually run out of PIDs
  - Prevention: The administrator should limit the number of processes a user can spawn
- To allow a zombie to die, its parent must explicitly "wait" for its children to finish
  - It can block until one of its children has died
  - Or it can setup a signal handler for the SIGCHLD signal to be asynchronously notified

Waiting for Children Processes [1/2]

- To block until some child process completes:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

- If a child process has terminated, store the exit status in *status and continue
- Otherwise, block until any child terminates, and then store its status
  - If status is NULL , don't store exit status
- wait returns the PID of the child process that has terminated, or -1 on error
- A child can terminate normally or be killed by a signal
- WIFEXITED tells you if the child terminated normally; use WEXITSTATUS to get the exit status

wait Example

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(void)
{
  pid_t child;
  int status, exit_status;
  if ((child = fork()) == 0) {
      sleep(5);
      exit(8);
  }
  wait(&status);

if (WIFEXITED(status)) {
  exit_status = WEXITSTATUS(status);
  printf("Child %d done: %d\n", child, exit_status);
}

return 0;
}
```


Waiting for Children Processes [2/2]

- `waitpid()` gives you more control:

```
pid_t waitpid(pid_t pid, int *status, int option);
```

  - `pid` specifies which child to wait for (-1 means any, which is what wait does)
  - option=WNOHANG makes the call return immediately, if no child has already completed (otherwise use option=0)

- Example: a SIGCHLD signal handler

```
void sig_chld(int sig) {
  pid_t pid;
  int stat;
  while ( (pid=waitpid(-1, &stat, WNOHANG)) > 0 ) {
    printf("Child %d exited with status %d\n", pid, stat);
  }
  signal(sig, sig_chld);
}
```

Orphan Processes

- What if a parent dies without waiting for all of its children?
- These unwaited-for children are called **orphan processes**
  - Any orphan processes are adopted by a process called init (`PID 1`)
  - `init` calls `wait` on them so that they can terminate and be cleaned up
- If a long-living parent keeps creating processes but doesn't wait on them, the process table will eventually get full

The exec Family

- There are six functions whose names start with exec that are used to overwrite the current process image with a new program
- They differ only in how they are called, not what they do.

```
int execl(const char *path, const char *arg, ..., (char*) NULL);
```

  - The first parameter, `path`, is the executable le to run
  - Then come the command line parameters to pass to `path`
    - When path runs, those parameters are in `argv[0], argv[1], ...`
- If successful, the `exec` functions do not return (The original program is gone!).

Example

```
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>

int main(void)
{
  printf("Before exec\n");
  execl("/bin/ls", "ls", "-l", (char *)NULL);
  perror("execl");
  exit(1);
}
```


The exec() System Call [2/4]

- `fork()` and `exec()` could be used separately
  – Imagine examples when this could happen?
- But most commonly they are used together:

```
if (fork()==0)     /* child */
{
  exec("hello");  /* load & execute new program */
  perror("Error calling exec()!\n");
  exit(1);
}
else  /* parent */
{
  ...
}
```

Other exec Functions

- execv
  - Takes an array of strings instead of multiple arguments
  - More convenient for building commands at runtime
- execlp, execvp
  - Search path (i.e. do not have to provide full path to executable
- execle, execve
  - Accept environment variables

The exec() System Call [4/4]

- There are 4 versions of exec(), depending on 2 criteria
  - Search for program in (i) absolute path, or (ii) $PATH
  - Command-line arguments passed as list or single vector (array)
- execl() example:

```
execl("/bin/ls", "ls", "-l", "/home/spyros", NULL);
```

	absolute path	$PATH
list (*n+1* parameters)	`execl()`	`execlp()`
vector (2 parameters)	`execv()`	`execvp()`

- execv() example:

```
char *params[4] = {"ls", "-l", "/home/rony", NULL};
execv("/bin/ls", params);
```

Questions

- Q1: What happens when you type a command in the shell?
  - The shell process forks (so, makes a copy of itself)
  - The child process execs the required program
- Q2: What if the command contains pipes?
  - `.:ls | grep ".txt" | wc`
    - The shell forks once per process
    - Creates pipes and arranges output of Pi to be input of Pi+1
- Q3: How does a process end?

Shell Skeleton

```
while (1) // Infinite loop
{
     print_prompt();
     read_command(command, parameters);
     if (fork()) { //Parent
          wait(&status);
     }
     else {
          execve(command, parameters, NULL);
     }
 }
```

Inter-Process Communication

- After a fork, we have two independent processes
- They have separate address spaces (most importantly, separate copies of variables)
- So, the processes can't use variables to communicate
- They could communicate using files, but coordination is difficult

Inter-Process Communication

- By default, processes cannot influence each other
  - Executed in isolation
  - Different address spaces
- There are 4 mechanisms for processes of the same computer to communicate
  - Signals: Send a signal to another process (SIGINT, SIGKILL, etc.)
  - Pipes: Communication channel to transfer data
  - Shared memory: the same memory area accessible to multiple processes
  - Semaphores: perform synchronization (e.g., to regulate access to shared memory)
- All these IPC methods work only between processes of the same computer!

Stopping a Process

- A process stops (without error) when:
  - The main() function returns
- or
  - the program calls exit()

Stopping a Process

- No process can directly kill another process, not even the kernel!
  - It can only send a signal to it asking it to terminate itself
  - The signal will invoke the appropriate **signal handler** (a function)

Signals

- ctrl+c is the simplest form of a signal, typically used to to terminate a program, what really happens?
  - Kernel is told by the terminal driver that you've hit the interrupt character, and sends SIGINT to the process
  - By default, `SIGINT` terminates the process
  - Signal `SIGKILL` has the same effect
    - But it cannot be overwritten
- Similarly: ctrl+z sends a SIGTSTP to a process
  - By default, `SIGTSTP` stops the process
  - There is also a `SIGSTOP` , which is guaranteed to stop a process (the process can't prevent it)
- The kernel sends several other signals to terminate processes when bad things happen, such as:
  - `SIGBUS`: hardware fault
  - `SIGFPE`: floating-point error
  - `SIGILL`: illegal instruction
- There are also two special signals, `SIGUSR1` and `SIGUSR2`
  - We can use these for our own purposes (kernel doesn't use them)
    - By default, they terminate the process

Sending Signals from the Shell

- Use kill -SIGNAME pid... to send signal SIGNAME to one or more processes
  - e.g. kill -SIGINT 11248
  - e.g. kill -SIGKILL 11248
- Unlike with SIGINT , no program can do anything about a SIGKILL

Ignoring and Catching Signals

- Signals are unexpected, asynchronous events: they can happen at any time
- Unless you make special arrangements, most signals terminate your process
- Three options
  - Write a signal handler function (called automatically upon the receipt of the signal), or
  - Ignore the signal (the signal does nothing to your process), or
  - Use the default action
  - `sigaction` is used for all of these purposes

`sigaction`

```
int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);
```

- sig: signal whose handling you want to change (e.g. SIGINT)
- act: set its sa_handler field to
  - The address of a handler function (i.e. a function pointer), or
  - SIG_IGN (ignore signal), or
  - SIG_DFL (default action)
- oldact: if not NULL, it will hold the handler information for signal sig before this sigaction call changes it

sigaction (sigaction.c)

```
int i = 0;
void handler(int signo) {
    fprintf(stderr, "Sig %d; total %d.\n", signo, ++i);
}

int main(void) {
    struct sigaction newact;
    sigemptyset (&newact.sa_mask); newact.sa_flags = 0;
    newact.sa_handler = handler;
    if (sigaction(SIGINT,&newact, NULL) == -1)
      exit(1);
    if (sigaction(SIGTSTP,&newact, NULL) == -1)
      exit(1);
    for(;;);      //Infinite loop
}
```


Signal Handling [1/3]: Sending Signals Between Processes

- Signals can be sent by one process (including a kernel process) to another
  - SIGSEGV: segmentation fault (non-authorized memory access)
  - SIGBUS: bus error (non-aligned memory access)
  - SIGPIPE: you tried to write in a pipe with no reader
  - SIGCHLD: one of your children processes has stopped
  - SIGSTOP, SIGCONT: pause and continue a process
  - SIGUSR1, SIGUSR2: two generic signals to be used by user programs
  - You can get a complete list of signals with `kill -l`

- Signals can also be sent explicitly: kill()

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- **This function does not necessarily kill processes!**

Signal Handling [2/3]

- To setup a custom signal handler:

```
#include <signal.h>
void (*signal(int signum, void (*sighandler)(int)))(int);
```

informally: `HANDLER *signal(int signum, HANDLER *sighandler);`
where HANDLER would be a function like this: `void sighandler(int)`

- **Example:**

```
#include <signal.h>

void myhandler(int sig) {
  printf("I received signal number %d\n", sig);
  signal(sig, myhandler);    /* set it again it */
}

int main() {
  void (*oldhandler)(int);
  oldhandler = signal(SIGINT, myhandler);
  signal(SIGUSR1, myhandler);
  ...
}
```


Signal Handling [3/3]

- When a signal handler is run, the signal gets associated back to its default behavior
  - You must call `signal()` again in the signal handler
- Attention: Unix normally does not queue signals
  - If the same signal is sent multiple times to the same process at short time intervals, **the signal may be delivered only once**
- Recent Unix systems have other signal handling calls that queue signals
  - We will not study them here

Signal Handling [3/3]

- When a signal handler is run, the signal gets associated back to its default behavior
  - You must call `signal()` again in the signal handler
- Attention: Unix normally does not queue signals
  - If the same signal is sent multiple times to the same process at short time intervals, **the signal may be delivered only once**
- Recent Unix systems have other signal handling calls that queue signals
  - We will not study them here

Pipes

- Pipe: one-way, first-in first-out communications channel
- Usually, one process writes into a pipe, and another process reads from the pipe

```
int pipe(int fds[2]);
```

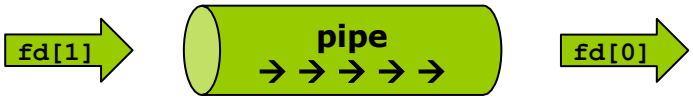
- You pass a pointer to two integers (i.e. an array or a malloc of two integers), and pipe fills it with two newly-opened FDs

Pipes [1/3]

- A pipe is a unidirectional communication channel between processes
  - Write data on one end – Read it at the other end
  - Bidirectional communication? Use TWO pipes.
- Creating a pipe:

```
#include <unistd.h>
int pipe(int fd[2]);
```

- The return parameters are:
  - `fd[0]` is the file descriptor for reading
  - `fd[1]` is the file descriptor for writing



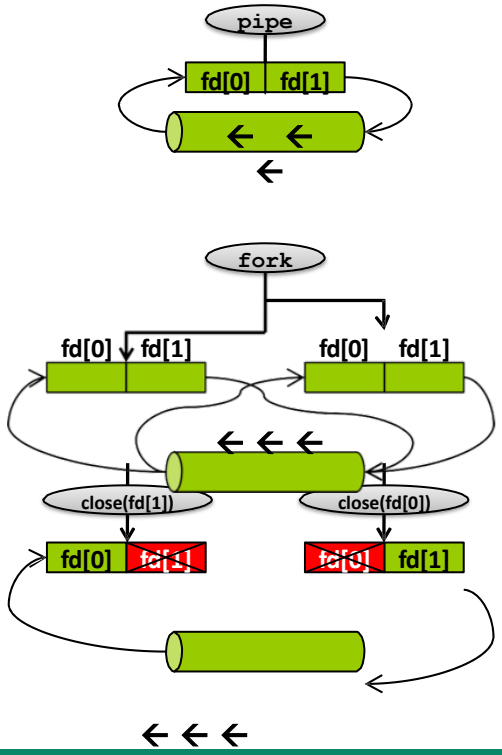
Example

- One process talking to itself through a pipe:

```
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#define MSG_SIZE 13
char *msg = "hello, world\n";

int main(void) {
   char buf[MSG_SIZE];
   int p[2];
   if (pipe(p) == -1) {
   perror("pipe");
   exit(1);
   }

write(p[1], msg, MSG_SIZE);
read(p[0], buf, MSG_SIZE);
write(STDOUT_FILENO, buf, MSG_SIZE);
return 0;
}
```

Pipes [2/3]: Pipes are often used in combination with fork()

```
int main() {
  int pid, fd[2]; char
  buf[64];

  if (pipe(fd)<0) {
  perror("pipe");
  exit(1);
  }
  pid = fork();
  if (pid==0)        /* child */
  {
    close(fd[0]);    /* close reader */
    write(fd[1],"hello, world!",14);
  }
  else {             /* parent */
  {
    close(fd[1]);    /* close writer */
    if (read(fd[0],buf,64) > 0)
      printf("Received: %s\n", buf);
    waitpid(pid,NULL,0);
   }
 }
```



Pipes [3/3]

- Pipes are used for example for shell commands like:

```
sort foo | uniq | wc
```

- Pipes can only link processes which have a common ancestor
  - Because children processes inherit the file descriptors from their parent

- What happens when two processes with no common ancestor want to communicate?
  - Named pipes (also called FIFO)
  - A special file behaves like a pipe
  - Assuming both processes agree on the pipe's filename, they can communicate
    - mkfifo() for creating a named pipe
    - open(), read(), write(), close()

Shared Memory [1/5]

- Shared memory allows multiple processes to have direct access to the same memory area
  - They can interact through the shared memory segment
- Shared memory segments must be created, then attached to a process to be usable. Then, you must detach and destroy them.
- When using shared memory, you must be very careful about race conditions, and solve them using semaphores.

Shared Memory [2/5]

- To create a shared memory segment:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

  - **key**: rendezvous point (key=IPC_PRIVATE if it will be used by children processes)
  - **size**: size of the segment in bytes
  - **shmflg**: options (access control mask)
  - **Return value**: a shm identifier (or -1 for error)

Shared Memory [3/5]

- To attach a shared memory segment:

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- **shmid**: shared memory identifier (returned by shmget)
- **shmaddr**: address where to attach the segment, or NULL if you don't care
- **shmflg**: options (access control mask)
  - SHM_RDONLY: read-only

```
int shmdt(const void *shmaddr);
```

- To detach a shared memory segment:
  - shmaddr: segment address
  - Attention: shmdt() does not destroy the segment!

Shared Memory [4/5]

- To destroy a shared memory segment:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- shmid: shared memory identifier (returned by shmget)
- cmd=IPC_RMID to destroy the segment
- buf: NULL as far as we are concerned

- Shared memory segments stay persistent even after all processes have died!
  - You must destroy them in your programs
  - The ipcs command shows existing segments (and semaphores)
  - You can destroy them by hand with: ipcrm shm <id>

Shared Memory [5/5]: Example

```
int main() {
  int shmid = shmget(IPC_PRIVATE, sizeof(int), 0600);
  int *shared_int = (int *) shmat(shmid, 0, 0);
  *shared_int = 42;

  if (fork()==0) {
    printf("The value is:        *shared_int)
                %d\n",           ;
    *shared_int = 12;
  } shmdt((void *) shared_int);
  else {
    sleep(1);
    printf("The value is: %d\n", *shared_int);
    shmdt((void *) shared_int);
    shmctl(shmid, IPC_RMID, 0);
  }
}
```

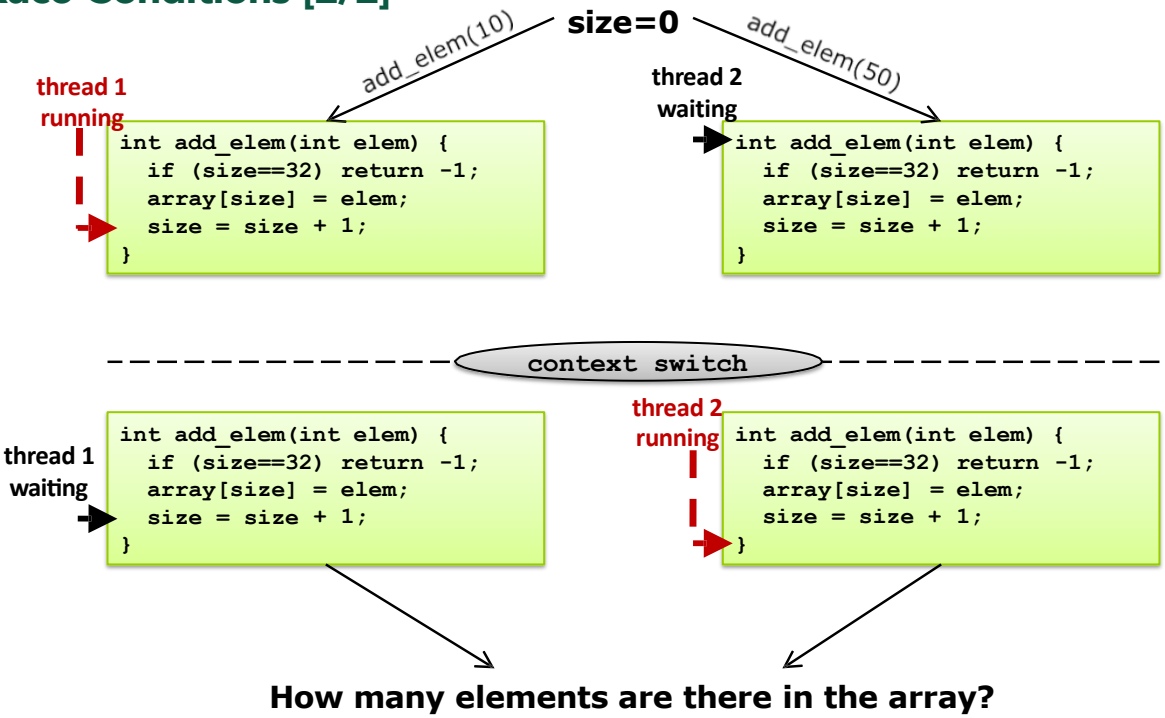

Race Conditions [1/2]

- When several processes share a resource, you must always wonder
- if synchronization is needed
- Example: Add an element to the end of an array

```
int array[32], size;

int add_elem(int elem)
{
  if (size==32) return -1;
  array[size] = elem;
  size = size + 1;
}
```

- This is a race condition: if two processes execute the function simultaneously, the result can be wrong

Race Conditions [2/2]



How many elements are there in the array?

Semaphores [1/6]

- In many cases you need to synchronize processes
  - When they share a resource (shared memory, file descriptor, device, etc.)
  - When a process needs to wait for a given event
- Semaphores are positive integers with two methods: UP() and DOWN()
  - DOWN():
    - If sem>=1 then sem=sem-1
    - Otherwise, block the process
  - UP():
    - If there are blocked processes, then unblock one of them
    - Otherwise, sem=sem+1
- Semaphore operations are atomic

Semaphores [2/6]

- Semaphores are generally used for two goals:
  - Mutual exclusion: only one process at a time can be within a section of code
    - Start with sem=1
    - To enter the mutex section: DOWN(sem)
    - To leave the mutex section: UP(sem)
  - Process synchronization: wait for a given event
    - Start with sem=0
    - To wait for the event: DOWN(sem)
    - To trigger the event: UP(sem)
    - What happens if the event is triggered before the other process waits for it?
- QUESTION: How can you solve the race condition in add_elem()?
- QUESTION: What if a critical region should accept up to K>1 processes?

Semaphores [3/6]

- Semaphores are created in arrays

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

- To create an array of semaphores:
  - key: rendezvous point or IPC_PRIVATE
  - nsems: number of semaphores to create
  - semflg: access rights
  - Return value: a semaphore array identifier

Semaphores [4/6]

- UP() and DOWN() are realized with the same function:
  - semid: the semaphore array identifier
  - sops: an array of commands to be issued
  - nsops: the size of sops

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

- Here's the sembuf struct:

```
struct sembuf {
   short  sem_num;   /* semaphore  number: 0 = first */
   short  sem_op;    /* semaphore  operation */
   short  sem_flg;   /* operation  flags */
};
```

- sem_op: 1 for UP(), -1 for DOWN()
- All operations are executed together atomically as a whole

Semaphores [5/6]

- To manipulate a semaphore:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

  - semid: the semaphore array identifier
  - semnum: the semaphore number
  - cmd: command
    - IPC_RMID: destroy the semaphore
    - GETVAL: return the value of the semaphore
- Like shared memory segments, semaphores persist in the system after all processes have completed
  - You must destroy them in your programs
  - By hand: ipcs, ipcrm

Semaphores [6/6]

A meaningless example

```
int main() {
  struct sembuf up   = {0,  1, 0};
  struct sembuf down = {0, -1, 0};
  int my_sem, v1, v2, v3, v4;

  my_sem = semget(IPC_PRIVATE, 1, 0600); /* create semaphore */
  v1 = semctl(my_sem, 0, GETVAL);

  semop(my_sem, &up, 1);                 /* UP() */
  v2 = semctl(my_sem, 0, GETVAL);

  semop(my_sem, &down, 1);               /* DOWN() */
  v3 = semctl(my_sem, 0, GETVAL);

  semctl(my_sem, 0, IPC_RMID);           /* destroy */
  v4 = semctl(my_sem, 0, GETVAL);

  printf("Semaphore values: %d %d %d %d\n", v1, v2, v3, v4);
}
```


Posix Threads

Threads vs. Processes

- Multi-process programs are expensive:
  - fork() needs to copy all the process' memory, etc.
  - interprocess communication is hard

- Threads: "lightweight processes"
  - One process contains several "threads of execution"
  - All threads execute the same program (but can be at different stages within it)
  - All threads share process instructions, global memory, open files, and
- signal handlers
  - Each thread has its own thread ID, program counter (PC), stack and
- stack pointer (SP), errno, and signal mask
  - There are special synchronization primitives between threads of the same process

Threads in C and Java

- Posix Threads
  - Posix Threads (pthreads) are standard among Unix systems
    - Also available on Windows through 3rd party libraries (Pthreads-w32)
  - The operating system must have special support for threads
    - Linux, Solaris, and virtually all Unix systems have it
  - Programs must be linked with –lpthread
    - Beware: Solaris will compile fine even if you forget the –lpthread (but your program will not work)

- Java Threads
  - Threads are a native feature of Java: every virtual machine has thread support
  - They are portable on any Java platform
  - Java threads can be:
    - mapped to operating system threads (kernel threads or native threads)
    - or emulated in user space (user threads or green threads)

Creating a pthread

- To create a pthread:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

  - thread: thread id (this is a return argument)
  - attr: attributes (i.e., options)
  - start_routine: function that the thread will execute
  - arg: parameter to be passed to the thread
  - Return value: 0 on success, error value on failure

- To initialize and destroy the default pthread attributes

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```


Stopping a pthread

- A pthread stops when:
  - Its process stops
  - Its parent thread stops
  - Its start_routine() function returns
  - It calls pthread_exit:

```
#include <pthread.h>
void pthread_exit(void *retval);
```

- Like processes, stopped threads must be waited for:

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

pthread Create/Delete Example

- To create a pthread:

```
#include <pthread.h>

void *func(void *param) {
  int *p = (int *) param;
  printf("New thread: param=%d\n",*p);
  return NULL;
}

int main() {
  pthread_t id;
  pthread_attr_t attr;
  int x = 42;

  pthread_attr_init(&attr);
  pthread_create(&id, &attr, func, (void *) &x);
  pthread_join(id, NULL);
}
```


Detached Threads

- A "detached" thread:
  - does not need to be joined by pthread_join()
  - does not stop when its parent thread stops
- By default, threads are "joinable" (i.e. "attached")
- To create a detached thread, set an attribute before creating the thread:

```
pthread_t id;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&id, &attr, func, NULL);
```

- You can also detach a thread later with pthread_detach()
  - But you cannot reattach it!

Race Conditions with Threads

- Threads share most resources by default
  - memory , file descriptors, etc.
- Attention! Danger!
  - It is very easy to make race conditions without even noticing
- You must always wonder if synchronization is needed
  - And solve them with special thread synchronization primitives
- Pthreads have two synchronization concepts:
  - Mutex
  - Condition Variables

Pthread Sync with Mutex [1/2]

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Pthread Sync with Mutex [2/2]

```
pthread_mutex_t mutex; int

add_elem(int elem) {
   int n;
   pthread_mutex_lock(&mutex);
   if (size==32) {
      pthread_mutex_unlock(&mutex);
      return -1;
   }
   array[size++] = elem; n
   = size;
   pthread_mutex_unlock(&mutex);
}

int main() {  pthread_mutexattr_t
   attr;
   pthread_mutexattr_init(&attr);
   pthread_mutex_init(&mutex, &attr);
   ...
   pthread_mutex_destroy(&mutex);
}
```


Thread Safety with Unix Primitives

- Attention: Some Unix primitives and library functions have similar internal race conditions
  - Many primitives were not designed with threads in mind
  - When writing a multi-threaded program, you must always check in the man pages if library functions are thread-safe
- Example:
  - gethostbyname() is not thread-safe
- You can still use it in multi-threaded programs
  - but only within a mutex protection

Pthread Sync with Condition Variables

- Condition Variables
  - One thread waits for an event triggered by another thread

- Condition variables always work together with a mutex and a predicate variable (e.g., a boolean, an int, etc.)
  - The predicate variable is used to make sure the event has really been triggered
    - Sometimes threads are woken up without the event having been triggered!
  - The mutex is used to synchronize access to the condition variable and the predicate variable

- Details are out of the scope of Networks Programming
  - You can find a detailed explanation at: http://www.ibm.com/developerworks/linux/library/l-posix3/ https://computing.llnl.gov/tutorials/pthreads/

API for Condition Variables

- At init time:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var; int
predicate = FALSE;

pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_var, NULL);
```

- To wait for an event:

```
pthread_mutex_lock(&mutex);
while (predicate==FALSE)
   pthread_cond_wait(&cond_var,&mutex);
predicate = FALSE;
pthread_mutex_unlock(&mutex);
```

- To trigger an event:

```
pthread_mutex_lock(&mutex);
predicate = TRUE;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Java Threads

Creating Java Threads

- A thread is a class that inherits from Thread
- You must overload its run() method:

```
public class MyThread extends Thread {
  private int argument;

  MyThread(int arg) {
    argument = arg;
  }

  public void run() {
    System.out.println("New thread started! arg=" + argument);
  }
}
```

- To start the thread:

```
MyThread t = new MyThread(1050);
t.start();
```

Stopping Java Threads

- A thread stops when its run() method returns
- You do not need to join() for a Java thread to finish
  - But you can, if you want:

```
MyThread t = new MyThread(1050);
t.start();
...
t.join();
```


Synchronization with Monitors [1/2]

- A monitor is similar to a mutex:

```
public class AnotherClass {
  synchronized public void methodOne()          {  ...  }
  synchronized public void methodTwo()          {  ...  }
  public void methodThree()                     {  ...  }
}
```

- Each object contains one mutex, which is locked when entering a synchronized method and unlocked when leaving
- Locking is at the object (instance) level
  - Two threads cannot be executing synchronized methods of a given object at the same time
    - But this is allowed by the same thread (e.g., one synchronized method calls another)
  - No restrictions apply to different objects of a given class

Synchronization with Monitors [2/2]

- So, the previous class:

```
public class AnotherClass {
  synchronized public void methodOne()          {  ...  }
  synchronized public void methodTwo()          {  ...  }
  public void methodThree()                     {  ...  }
}
```

…is equivalent to:

```
public class AnotherClass {
  private Mutex mutex;
  public void methodOne() { mutex.lock(); ...; mutex.unlock();}
  public void methodTwo() { mutex.lock(); ...; mutex.unlock();}
  public void methodThree() { ... }
```

…except that the Mutex class does not exist!

- QUESTION: Can you implement a Mutex class using monitors?

Condition Variables in Java [1/2]

- There is no real condition variable in Java
  - But you can explicitly block a thread
- All Java classes inherit from class Object the following:

```
class Object {
  void wait();       /* blocks the calling thread */
  void notify();     /* unblocks ONE thread blocked by this object*/
  void notifyAll(); /* unblocks ALL threads blocked by this object*/
}
```

  - wait(): causes current thread to wait until another thread invokes the
  - notify() or notifyAll() method for this object
  - notify(): wakes up a single thread that is waiting on this object's monitor
  - notifyAll(): wakes up all threads that are waiting on this object's monitor

Condition Variables in Java [2/2]

- This means, each object contains exactly one (and no more) condition variable
- The wait(), notify() and notifyAll() methods can only be called inside a monitor of that same instance. E.g.:

```
Integer myObj;  // or any other type of object

synchronized(myObj)
{
  myObj.wait();
}
```

- Questions:
  - [1/3] What if you need several condition variables in a given object?
  - [2/3] Can you implement a ConditionVariable class in Java?
  - [3/3] Can you implement a Mutex class in Java?