# Agenda – Advanced Signals

Handling the arrival of multiple, simultaneous signals

Blocking vs unblocking wait calls

Using signals to get more information about child state

Ignoring and delaying signals

Handling asynchrony: when a signal interrupts a signal handler

Tips for building your job shell

# Signals Review

What is a signal?

Name some common signals and when they occur

What command can we use to send a signal?

List three ways a process can respond to a signal.

How do process groups affect signal handling?

# Handling the arrival of multiple, simultaneous signals

Signals are not queued. If multiple signals arrive before the handler is called, the handler is still only called once

Pending signals are stored in a bit field, 1 bit for each type of signal

- Therefore, when a bit is one, it could be due to more than one occurrence

Example: Suppose a parent process creates multiple child processes that end at approximately the same time?

# Demo

```
void int_handler(int sig) {
    printf("Proc %d received signal %d\n”,getpid(), sig);
    exit(0);
}
void fork_with_sigint_handler() {
    pid_t pid, w_pid;
    int i, status;
    // install signal handler on SIGINT signals
    signal(SIGINT, int_handler);
    pid = fork();
    if (pid == 0) { // child
       while(1);
    } else { // parent
       kill(pid, SIGINT); // send child SIGINT signal
       w_pid = wait(&status);
    }
    exit(0);
}
```

In 1-2 sentences, describe what this program is doing.

# Demo – Draw the process timeline

```
int ccount = N; // a global variable

void child_handler(int sig){ // catch SIGCHILD
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    printf("signal %d from %d\n",sig, pid);
}
void forky(){ // fork N processes that sleep & exit
    pid_t pid[N];
    int i;
    signal(SIGCHLD, child_handler);// install SIGCHLD hndlr
    for (i = 0; i < N; i++){
       pid[i] = fork();
       if (pid[i] == 0) {
            sleep(1);
            exit(0);  // will send SIGCHLD to parent
       }
    }
    while (ccount > 0) {
          pause(); // Suspend parent until it gets a signal
    }
}
```

NOTE: this SIGCHLD handler is missing something for general handling

# Demo – Draw the process timeline

# More SIGCHILD signals can happen while we are handling this one*, so typically loop in handler until all have reaped all children:

```
void child_handler2(int sig) {
    int child_status;
    pid_t pid;
    while((pid = waitpid(-1,&child_status,WNOHANG))>0){
            ccount--;
            printf("signal %d from %d\n",sig, pid);
    }
}

void forky(){
    . . .
    signal(SIGCHLD, child_handler2);
    . . .
}
```

* Pending signals are stored in a bit vector, 1 bit for each type of signal:
    1: at least one instance of signal occurred (but there may have been more)

# Blocking vs Unblocking calls

```
pid_t pid = wait(&child_status); // blocks
```

```
pid = waitpid(-1,&child_status,WNOHANG) // non-blocking
```

# Getting detailed child information

waitpid tells us when a child has exited, but what if we want to know all changes in state, e.g. continued, stopped, running, exited?

```
pid = waitpid(-1, &status, WNOHANG | WUNTRACED | WCONTINUED)
```

Check the status to see what happened using if WIFSTOPPED, WIFCONTINUED, WIFEXITED, WIFSIGNALED, etc

# Handling asynchrony

**Watch Out!** Signals can arrive at any time

- Can interrupt a system call
- Can interrupt a signal handler

Example: A SIGCHILD signal might occur inside a SIGCHILD handler

If a handler performs a task that should not be interrupted (e.g. not **reentrant**), we should postpone signals within the handler

Can you think of some tasks that we would not want interrupted?

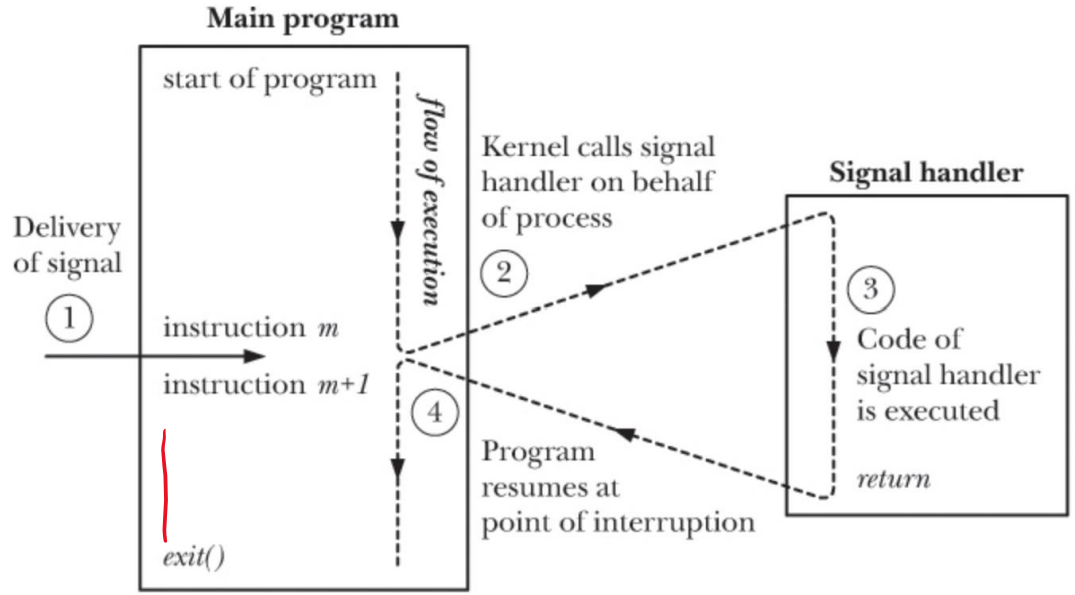



**Figure 20-1:** Signal delivery and handler execution

Linux Programming Interface

NOTE: Commands like printf, strsignal are not **reentrant** so can be "messed up" by multiple signals. [We will use them anyways since our programs are simple]

# Handling asynchrony: configuring your signal handler

Use **sigaction** to configure the behavior of signals when setting a handler

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

void handler(int sig)
{
  printf("Quitting!\n");
  exit(0);
}
```

```
int main()
{
  struct sigaction sa;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;
  sa.sa_handler = handler;
  if (sigaction(SIGINT, &sa, NULL) == -1)
  {
    perror("sigaction");
  }
  pause();
  return 0;
}
```

# Signals can interrupt system calls

Ex: A read from a file is interrupted by a signal

To specify that interrupted calls should be restart, set the corresponding flag

```
struct sigaction sa;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART ;
sa.sa_handler = handler;
checkerror(sigaction(SIGALRM, &sa, NULL));
```

# Signals can interrupt a handler

Solution: Block signals while executing the handler

- `sa_mask` can be used to specify signals that should be blocked during the handler
- sigprocmask can be used to modify the blocked signals for the process

When signals are blocked, using non-reentrant functions and global variables is safe

```
struct sigaction sa;
 //sigemptyset(&sa.sa_mask); // mask is empty – will still ignore own signals before re-calling handler
 sigfillset(&sa.sa_mask); // ignore all signals until handler function completes
 sa.sa_flags = SA_RESTART;
 sa.sa_handler = sig_alarm;
 checkerror(sigaction(SIGALRM, &sa, NULL));
```

# Working with sa_mask

```
struct sigaction sa;
sigfillset(& sa.sa_mask);
sigemptyset(& sa.sa_mask);
sigaddset(& sa.sa_mask, SIGTTIN);
sigdelset(& sa.sa_mask, SIGTTIN);
```

Visualize the above code

Exercise: Write code that ignores sig alarm within a SIGCHLD handler. What would the bitfield, sa_mask, look like?

# Demo - Alarm

```
struct timeval tstart;

void sig_alarm(int signum)
{
  printf("triggered...\n");
  sleep(5);
  struct timeval tend;
  gettimeofday(&tend, NULL);
  float dt = tend.tv_sec - tstart.tv_sec +
     (tend.tv_usec - tstart.tv_usec)/1.e6;
  printf("Process received alarm signal.”
      “Elapsed time = %.2f\n", dt);
}
```

NOTE: This demo also shows that signals are not queued!

```
int main()
{
  struct sigaction sa;
  sigemptyset(&sa.sa_mask);
  //sigfillset(&sa.sa_mask);
  sa.sa_flags = SA_RESTART;
  sa.sa_handler = sig_alarm;
  checkerror(sigaction(SIGALRM, &sa, NULL), "sigaction");

  struct itimerval v;  // sends alarm every 1 seconds
  v.it_interval.tv_sec = 1;
  v.it_interval.tv_usec = 0;
  v.it_value.tv_sec = 1;
  v.it_value.tv_usec = 0;
  checkerror(setitimer(ITIMER_REAL, &v, NULL), "settimer");

  gettimeofday(&tstart, NULL);
  while(1) pause();
  return 0;
}
```

# Ignoring signals

A signal may be blocked, which means that it will not be delivered until it is later unblocked.

Between the time when it is generated and when it is delivered a signal is said to be pending.

Each thread in a process has an independent signal mask, which indicates the set of signals that the thread is currently blocking.

- **pthread_sigmask**
- **sigprocmask**

A child created via **fork** inherits a copy of its parent's signal mask; the signal mask is preserved across **execve**

# Example: Ignoring signals for a process

sigprocmask can be used to block/unblock signals for a process. For example, to ignore SIGTTOU and SIGTTIN, you would do

```
sigset_t newset, oldset;
sigaddset(&newset, SIGTTIN);
sigaddset(&newset, SIGTTOU);

sigprocmask(SIG_BLOCK, &newset, &oldset);
 // oldset can be used to restore the previous mask if desired
```

WATCH OUT: children inherit their parent’s signal masks

# Implementing job control in a shell: Version 1

When creating children with fork/exec, the children need to be in their own process group

If the child runs in the background, we only need to

- Put child in its own process group

If the child runs in the foreground, we need to

- Put child in its own process group
- Give the child the terminal
- Wait for it finish
- Take back the terminal

# Foreground vs background

```
void handler(int sig) {
  printf("%d got the signal!\n", getpid());
}
```

```
int main(int argc, char* argv[])
{
  bool background = true;

  signal(SIGINT, handler);
  int pid = fork();
  if (pid == 0)
  {
    setpgid(getpid(), getpid());
    sleep(5);
    printf("Child is exiting\n");
  }
```

```
else  {
    setpgid(pid, pid); // put child in their own group
    if (!background)    {
      printf("Running %d (child) in foreground.\n", pid);
      printf("Running %d (parent) in background.\n", getpid());

      checkerror(tcsetpgrp (STDIN_FILENO, pid), "child control");
      checkerror(waitpid(pid, NULL, 0), "wait child");
    }
    else    {
      printf("Running %d (child) in background.\n", pid);
      printf("Running %d (parent) in foreground.\n", getpid());
    }
    sleep(5);
    printf("Parent is exiting\n");
  }
  return 0;
}
```

# Implementing job control: Version 2

How to handling switching between the foreground and background?

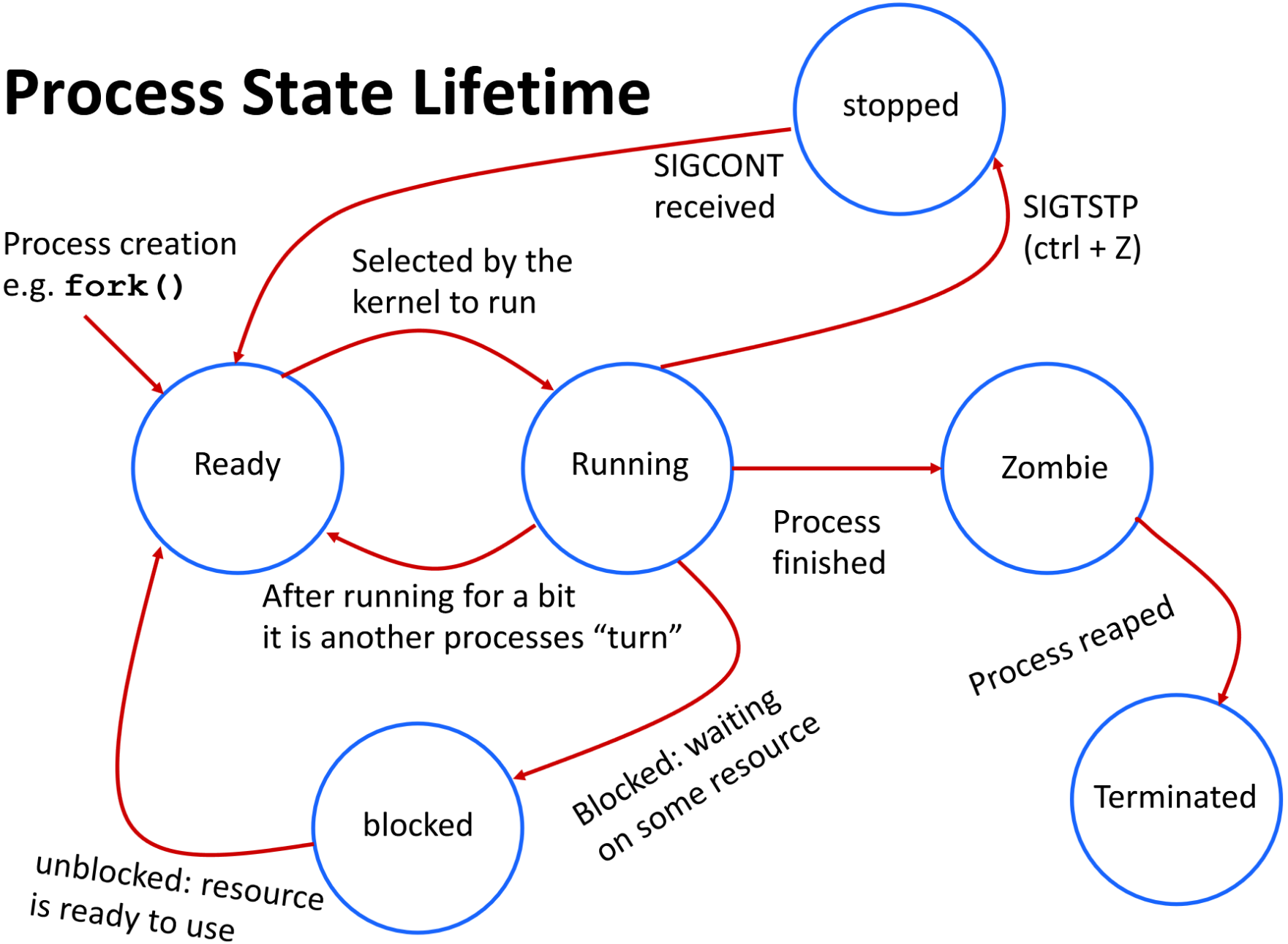
When a process gets the signal SIGSTOP, the process will not run on the CPU until it is resumed by the SIGCONT signal

Ex: Ctrl-Z

Ex: illegal use of terminal generates SIGTSTP/SIGTTOU

When the user resumes a process (e.g. with fg/bg), we need to send the SIGCONT signal

# Process State Lifetime



# Job control example: one parent, one child

Recommended setup:

- Parent should ignore SIGTTOU and SIGTTIN so it can change the terminal ownership when child processes stop or exit
- Child signal handler should ignore all signals so that bookkeeping and reporting is safe

Demo: A simple fork program where the child runs in the background to start

Run with `strace -e 'trace=!all' -I 3 -D -f ./fgbg`

# Scenario: Uses does fg

The background process is moved to the foreground

# Scenario: Uses does Ctrl-Z

The foreground process is stopped (SIGTSTP)

# Scenario: Uses does bg

The suspended process is resumed in the background

# Scenario: Uses does quit

The suspended process is resumed in the background